# Procedures and design by contract

## Procedural abstraction

A procedure (or subroutine) is a named piece of code.

Typically we abstract away from the details of **how** a procedure accomplishes its goal and focus on **what** that goal is.

The tool for doing such abstraction is the procedure's contract.

## Contracts for procedures

Consider an algorithm to find a minimum spanning Forest of a graph where the edges are $\{e_0, ..e_m\}$ and each edge $e$ has a weight of $w(e)$.

Input a graph $G$ with nodes $N = \{u_0, ..u_n\}$ edges $E = \{e_0, ..e_m\}$ and a real function $w$ defined on all items of $E$.

Output a $F$ subset of $E$ that forms a minimum weight spanning forest of the graph

Method

var $a :=$ new array $\langle \text{Edge} \rangle$ $(m)$

for( $i \leftarrow [0, ..m]$ ) do $a(i) := e_i$ end for

sort $a$ by weight so that the least weight edges are at the front

$F := \emptyset$

for( $i \leftarrow [0, ..m]$ ) do

if $F \cup \{a(i)\}$ has no cycle then

$F := F \cup \{a(i)\}$ end if end for

We need to sort an array of edges

The contract is

procedure *edgeSort*(var $a : \text{array}\ \langle \text{Edge} \rangle$ , $w : \text{Edge} \xrightarrow{\text{par}} \mathbb{R}$)
  precondition $\forall i \in \{0, ..a.\text{length}\} \cdot a(i) \in \text{dom}(w)$
  changes $a$
  postcondition $a$ is a permutation of $a_0$
  $\wedge \quad \forall i, j \in \{0, ..a.\text{length}\} \cdot i \leq j \Rightarrow w(a(i)) \leq w(a(j))$

The contract specifies

- What must be true before each invocation (i.e. the precondition)
- Which variables (aside from local variables) may be changed by the procedure
- What the procedure guarantees about the state at the end of an invocation (the postcondition)

Like all contracts there are obligations and benefits. An obligation of one party generally corresponds to a benefit

to the other.

| | **Obligation** | **Benefit** |
|---|---|---|
| **Precondition** | The client's designer is obliged to ensure the precondition is true to start with. | The procedure's designer can assume the precondition is already true when it starts. |
| **Postcondition** | The procedure's designer is obliged to ensure the postcondition is true. | The client's designer can assume the postcondition is true after then invocation. |
| **Changes** | The procedure's designer is obliged not to change any other variables. | The client's designer can assume all other variables are left alone. |

If the client defaults on its obligations then the procedure need not respect its obligations.

Example

procedure *findMax*( $a : \text{array } \langle Int \rangle$ ) : *Int*

precondition $\exists i \in \{0, ..a.\text{length}\} \cdot a(i) \geq 0$

postcondition $\forall i \in \{0, ..a.\text{length}\} \cdot a(i) \leq a(result)$

The following is an acceptable algorithm according to the contract.

var $m := -1, r := 0, j := 0$
$\left\{ \begin{array}{l} \quad (\neg\,(a\,\{0,..j\} <^* 0) \Rightarrow a\,\{0,..j\} \leq^* a(r) = m) \\ \wedge\ (a\,\{0,..j\} <^* 0 \Rightarrow m \leq 0) \end{array} \right\}_1$
while $j < a.\,\mathrm{length}$
  if $a(j) > m$ then $m := a(j)\ r := j$ end if $j := j + 1$
end while
return $r$

As you can see, if one passes in the array $[-3, -2, -4]$, then the postcondition will not hold.

[Aside: Good programming practice suggests that, if the precondition is false, the code should —if practical— alert the programmer somehow. E.g. by throwing an exception. This is an example of **defensive programming**. Defensive programming dictates that code should check for errors —internal or environmental— whenever practical. However I am not going to put this behaviour into the contract, because it is not a behaviour I want the client to be able to depend on. We leave this as a matter of pragmatics rather than semantics. A good designer will weigh the probability of an error being caught and the benefit that will provide vs the costs of checking for an error. End of aside.]

[1] The notation $a\{0,..j\} <^* 0$ means all items of $a$ with indeces in $\{0,..j\}$ are less than 0

Conventions for procedure pre- and postconditions.

- In postconditions we use $v_0$ to mean "the initial value of $v$" (the value at the start of the invocation).
- In effect we have (for all variables) $v_0 = v$ as an implicit conjunct of the precondition.
- If a global variable $v$ is **not** mentioned in the "changes" clause, its final value should be the same as it's initial value.
- In effect we have (for these variables) $v = v_0$ in the postcondition.

For example

procedure *sort*( var $a : \text{array } \langle \text{Edge} \rangle$ , $w : \text{Edge} \xrightarrow{\text{par}} \mathbb{R}$ )
precondition $\forall i \in \{0, ..a.\text{length}\} \cdot a(i) \in \text{dom}(w)$
changes $a$
postcondition $a$ is a permutation of $a_0$
$\wedge \forall i, j \in \{0, ..a.\text{length}\} \cdot i \leq j \Rightarrow w_0(a(i)) \leq w_0(a(j))$

written without these conventions is

procedure *sort*( var $a : \text{array } \langle \text{Edge} \rangle$ , $w : \text{Edge} \xrightarrow{\text{par}} \mathbb{R}$ )
precondition $(\forall i \in \{0, ..a.\text{length}\} \cdot a(i) \in \text{dom}(w))$
$\wedge\ a_0 = a \wedge w_0 = w \wedge x_0 = x \wedge y_0 = y \wedge ...$
postcondition $a$ is a permutation of $a_0$
$\wedge \forall i, j \in \{0, ..a.\text{length}\} \cdot i \leq j \Rightarrow w_0(a(i)) \leq w_0(a(j))$
$\wedge\ x_0 = x \wedge y_0 = y \wedge ...$

# Recursion

Recursive procedures are ones that may call themselves directly or indirectly.

Thinking about what a procedure does, rather than how it does it (procedural abstraction) is the way to deal with recursive procedures.

For example, consider the sorting a deck of 52 cards.

We'll assume there is some partial order $\leq$ on the cards.

For example $(s_0, v_0) \leq (s_1, v_0)$ could be defined by

$$((s_0, v_0) \leq (s_1, v_0)) = (s_0 < s1 \vee (s_0 = s_1 \wedge v_0 \leq v_1))$$

where $\clubsuit < \diamondsuit < \heartsuit < \spadesuit$. or it could be defined by

$$((s_0, v_0) \leq (s_1, v_0)) = (v_0 \leq v_1)$$

Here is an algorithm for sorting cards

- Pick any card. Call it $x$. Remove it from the deck.
- Make a pile $A$ of all remaining cards $< x$.
- Make another pile $B$ of all remaining cards $> x$.
- $x$ goes in neither pile, but other cards equal to $x$ can go in either pile.
- Ask a friend to sort pile $A$ and pile $B$
- Put $x$ on top of pile $B$ and pile $A$ on top of $x$.
- Now the deck is sorted.

It may appear that the task you are asking your friend to do is just as hard as the one you are trying to do, but it is not. You need to sort $52$ cards, they need to sort, at most, $51$.

Now let's do it with an array and allow duplicate values.

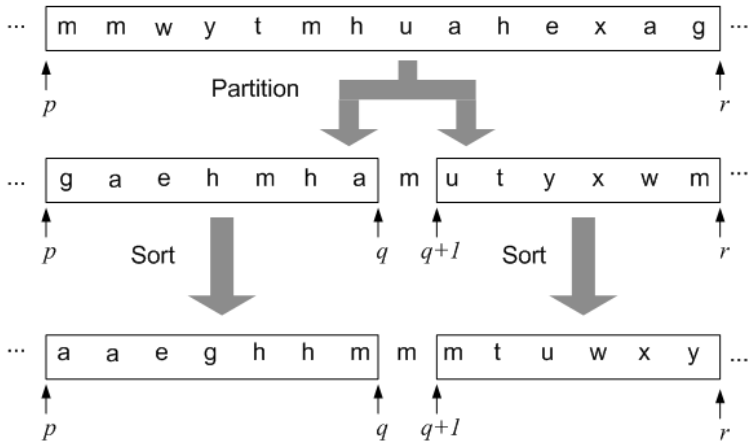
We'll assume that $x \leq y \wedge y \leq x \Rightarrow x = y$, for all $x, y \in T$.

Let's specify sorting a portion of an array with a contract

procedure *sort*( var $a : \text{array } \langle T \rangle$ ; $p, r : Int$ )
  precondition $0 \leq p \leq r \leq a.length$
  changes $a$, but only at indices $\{p, ..r\}$
  postcondition
    $a$ is a permutation of $a_0$
  $\wedge \quad \forall i, j \in \{p, ..r\} \cdot i \leq j \Rightarrow a(i) \leq a(j)$

Now without worrying about *how* our friends do their sorts, we can implement the sort specification with the following algorithm.

procedure *fairlyQuickSort*( var $a : \text{array } \langle T \rangle$ ; $p, r : Int$ )
implements sort$(a, p, r)$
  if $r - p > 1$ then
    val $i :=$ any value from $\{p, ..r\}$
    val $x := a(i)$
    var $q$
    *partition*( $a, p, r, x, q$ )
    { $p \leq q < r$
      and everything in $a\{p, ..q\}$ is $\leq x$
      and $a(q) = x$
      and everything in $a\,\{q + 1, ..r\}$ is $\geq x$ }
    sort( $a, p, q$ )
    sort( $a, q + 1, r$)
  end if
end *fairlyQuickSort*




This works assuming that partition permutes segment $p, ..r$ of $a$ such that the assertion is true, does not change any other items of $a$ and does not change $p$ or $r$.

So partition should implement the following contract

procedure *partition*$(\text{var } a : \text{array } \langle T \rangle;\ x : T;\ p, r : Int;\ \text{var } q : Int)$
precondition $p < r$ and $\{p, ..r\} \subseteq \text{dom}(a)$ and $x \in a\{p, ..r\}$
changes $a$ (but only permuting $a\,[p, ..r]$), $q$
postcondition $p \leq q < r$ and $a\{p, ..q\} \leq^* x$ and $x \leq^* a\{q+1, ..r\}$ and $x = a(q)$

where $S \leq^* x$ means $\forall y \in S \cdot y \leq x$ and similarly for $x \leq^* S$.

Now we can verify the partial correctness of $fairlyQuickSort$ based on the specifications of $sort$ and $partition$.

Exercise: Implement $partition$.

Of course our friends may use the same algorithm, and so may their friends, and so on. If all do, that gives the classic quick sort algorithm.

procedure *quickSort*( var $a : \text{array}\ \langle T \rangle\ ; p, r : Int$ )
implements sort$(a, p, r)$
  if $r - p > 1$ then
    val $i :=$ any value from $\{p, ..r\}$
    val $x := a(i)$
    var $q$
    *partition*( $a, p, r, x, q$ )
    { $p \leq q < r$
    $\wedge \quad a\{p, ..q\} \leq^* x \wedge a(q) = x \wedge a\,\{q+1, ..r\} \geq^* x$ }
    *quickSort*( $a, p, q$ )  *quickSort*( $a, q+1, r$)
  end if
end *quickSort*

For total correctness we need to show the algorithm will terminate.

For termination we can use a variant expression:

A **variant expression** for a recursive routine is an integer expression that

- can not be less than $0$ (assuming the precondition holds)
- is less (by at least 1) for each recursive invocation

With $quickSort$ a variant expression is $r - p$.

By the precondition, this is $\geq 0$.

The values of the variant for the recursive calls are, respectively, $q - p$ and $r - q - 1$.

Since $q < r$, we have $q - p < r - p$

Since $p \leq q$, we have $r - q - 1 < r - p$

There is no need to resort to proof by strong induction for each recursive subroutine. Just apply partial correctness and show there is a variant.

[Aside: Theory meets practice. Our $quickSort$ is perfect in theory. However in practice there is a problem. Real machines only approximate ideal machines. In particular each time a subroutine is invoked, some data needs to be added to the stack until the invocation ends. When I tried sorting an array of size $10^5$ in Java, I actually exceeded the stack limit of the JVM. We'll fix this problem later.]